Department of Physics and Astronomy
Heidelberg University

Bachelor Thesis in Physics
submitted by

Anna-Katharina Stsepankova

born in Mannheim (Germany)

2025

Model-Guided Parameterization for Flexible Operating Points on BrainScaleS-2

This Bachelor Thesis has been carried out by Anna-Katharina Stsepankova at the
Kirchhoff Institute for Physics in Heidelberg
under the supervision of
Prof. Johannes Schemmel

Abstract

BrainScaleS-2 is a mixed-signal neuromorphic platform that provides a substrate for experiments with spiking neural networks. Like other analog substrates, it is affected by fixed-pattern noise, which causes mismatches between identically designed components. Calibration mitigates this issue by determining the hardware configurations that yield the desired effective behavior. The current calibration library employs an iterative binary search algorithm to identify the operating point, a process that is time-consuming. This work presents a transformation-based parametrization approach that leverages an effective model to significantly accelerate calibration. By explicitly capturing both, hardware-model dependencies and interdependencies among hardware parameters, this method reduces the time needed to determine operating points by approximately an order of magnitude. Furthermore, the developed architecture enables unit translation from analog-to-digital converter (ADC) values to physical quantities, thereby providing interpretable readouts.

Zusammenfassung

BrainScaleS-2 bietet als neuromorphe Plattform zwar ein Substrat für Experimente mit neuronalen Netzen, leidet jedoch, wie jedes analoge Substrat, an durch den Herstellungsprozess verursachten zeitlich konstanten Variationen zwischen identisch entworfenen Komponenten. Kalibration kann helfen, dies einzuschränken, indem Hardware Parameter gefunden werden, die zum gewünschten Verhalten führen. Die aktuelle Kalibration verwendet ein iteratives Verfahren zur Operationspunktbestimmung, was ein zeitaufwendiger Prozess ist. Das Ziel dieser Arbeit ist es deshalb, ein transformationsbasiertes System zu entwickeln, welches die Parametrisierung signifikant beschleunigt. Dabei werden nicht nur Abhängigkeiten zwischen Modell- und Hardwareparametern berücksichtigt, sondern auch solche zwischen Hardwareparametern untereinander. Zudem wird diese Architektur verwendet, um eine Einheitsübersetzung von den Werten von Analog-Digitalwandlern zu physikalischen Einheiten umzusetzen.

Contents

1 Introduction

As various fields profit from the recent advancements in the field of artificial neural networks (ANN), the relevance and usage of artificial neural networks has become larger than ever [Schilling et al., 2020]. Nevertheless, this development gives rise to concerns about the sustainability and computational efficiency of this technology. For instance, common large language models can consume up to 40 Wh per prompt — about two orders of magnitude more energy than a Google search. [Jegham et al., 2025]. Recent research estimates the related carbon emissions to make up approximately 8% of global emissions in the following decade, with the energy demand of global data centers reaching 1000 TWh by 2026 [Wilhelm et al., 2025].
In contrast, the human brain, despite performing similar tasks, is much more energy efficient [Balasubramanian, 2021]. Therefore, spiking neural networks (SNN), that are inspired by the functionality of biological neurons, promise to improve the efficiency of neural networks, especially on specifically designed hardware [Bouvier et al., 2019]. SNNs encode the information in the timing of spikes, which makes them potentially more energy efficient than classic ANNs, because neurons only compute and communicate when a spike occurs, saving energy. This work utilizes the mixed-signal neuromorphic platform BrainScaleS-2 (BSS-2) [Pehle et al., 2022] that emulates SNNs. This system forms a working substrate for experiments and modelling, however, like in all analog substrates, fixed-pattern noise is unavoidable [Wunderlich et al., 2019]. This leads to mismatch between identically designed components [Weis, 2020], which can be reduced using calibration. The calibration process determines the hardware configuration that yields the desired analog behavior. The current calibration library utilizes an iterative binary search algorithm which optimizes the hardware parameters iteratively. However, this approach is time consuming due to the large amount of iterations.

An alternative would be to use a translation-based approach, where the model parameters describing the analog behavior are determined as a function of the hardware parameters, with the advantage being that once the transformation function has been determined no further measurements are necessary. The predecessor to BSS-2, BrainScaleS-1, as well as some BSS-2 prototypes, already employed such a system. Yet, its implementation only allowed for each model parameter to only depend on a single hardware parameter [Visions, 2013]. This assumption does not hold true in every case, since some hardware parameters on BSS-2 are interdependent [Hinterding, 2025]. The primary objective of this work is to develop a software architecture that implements a translation-based parametrization algorithm, considering interdependencies between hardware parameters. Measurements of exemplary parameters [Hinterding, 2025] are utilized to design and implement a functional Application Programming Interface (API) that allows to request the parametrization and serialization of arbitrary parameter constellations. This API is also used to additionally implement a unit translation from analog-to-digital converters (ADC) digital steps to SI units to achieve interpretable readouts.

2 Methods

2.1 Structure of the BrainScaleS-2 system

This section gives a basic description of the BSS-2 system architecture [Pehle et al., 2022], as shown in Figure 1.

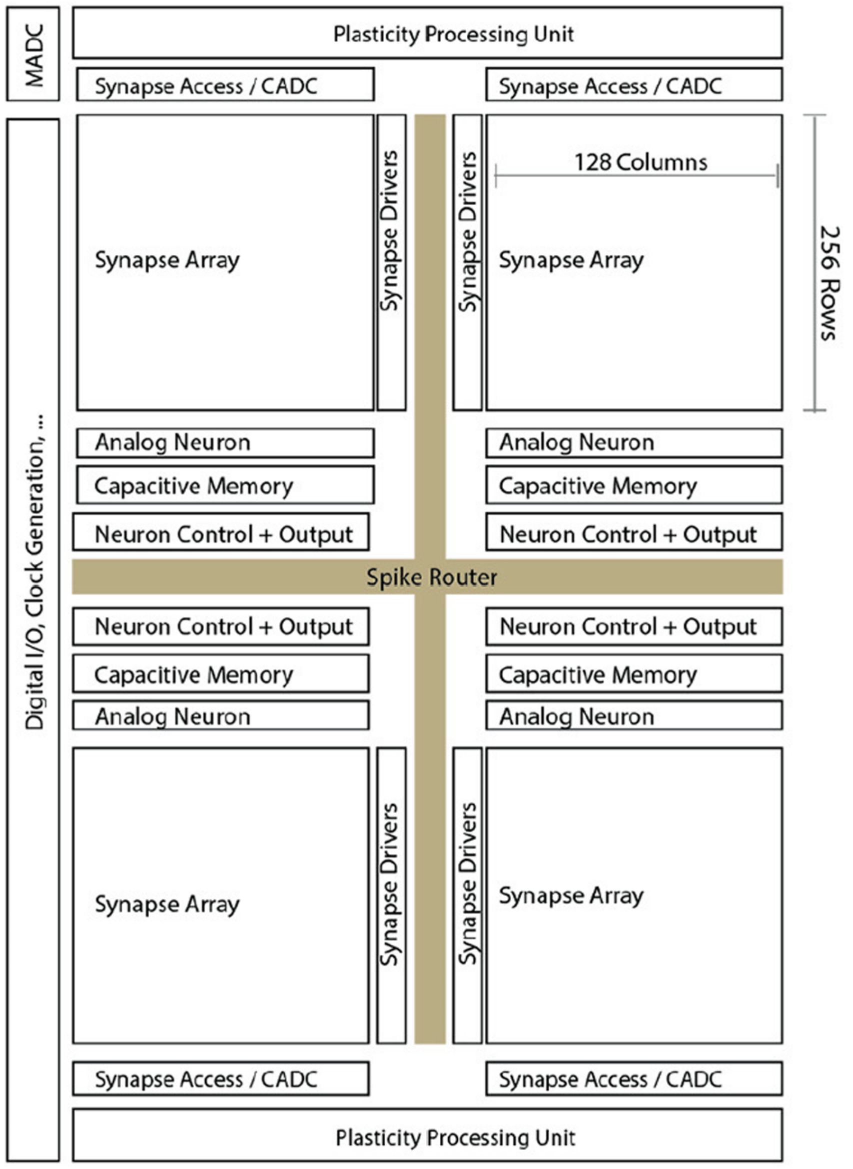


Figure 1: Overview of the HICANN-DLS-SR-HX chip structure. The chip is divided into four quadrants, each featuring a synapse array with 256 rows and 128 columns. Each column corresponds to a neuron circuit. The CADC makes analog observables accessible by the Plasticity Processing Unit (PPU). The upper and lower half of the chip each feature a PPU with CADC 512 channels each. There are four CADC units and one MADC unit per chip, totaling 1024 CADC channels and 2 MADC channels. Taken from [Pehle et al., 2022]

.

It is divided into four quadrants, each hosting a synaptic crossbar with 256 rows and 128 columns, corresponding to the neuron circuits. Additionally, each quadrant has its own Capacitive Memory (CapMem), which stores analog neuron parameters as well as global quadrant parameters. Dividing the space between the quadrants is the event-routing network, which is responsible for spike communication. There are two types of ADCs on the chip: the membrane analog-to-digital converter (MADC) and the columnar analog-to-digital converter (CADC). The MADC is featured once on every chip and has two channels, whereas the CADC is featured four times with 256 channels per quadrant, being 1024 channels in total. This enables the CADC to record the potential of all neurons at once. However, the MADC has a higher sampling frequency of 30MHz compared to the CADC's 1MHz.

2.2 Calibratable parameters

This section describes the model utilized by BrainScaleS-2 and gives an introduction to the calibratable parameters that are relevant for this work.

2.2.1 Leaky integrate-and-fire model

The HICANN-DLS-SR-HX chip uses a common simplification for spiking neurons, namely the adaptive exponential integrate-and-fire (AdEx) model [Brette R, 2005]. However, the measurements utilized in this work are based only on the leaky integrate-and-fire (LIF) part of the model [Schmidt, 2024].

The LIF model describes the temporal evolution of the membrane potential V(t) with the differential equation:

$$\tau_{mem} * \dot{V} = -[V(t) - V_{leak}] + \frac{I(t)}{g_{leak}} \tag{1}$$

where g_{leak} is the leak conductance, V_{leak} is the resting potential and I(t) denotes the synaptic input current. When the membrane potential reaches a threshold voltage, the neuron fires a spike, which is distributed to all following neurons. The membrane potential is clamped to the reset potential V_{reset}. The membrane time constant is given by $\tau_{mem} = \frac{C_m}{g_{leak}}$. Figure 2 depicts a conductance based circuit diagram emulating the model [Schmidt, 2024].

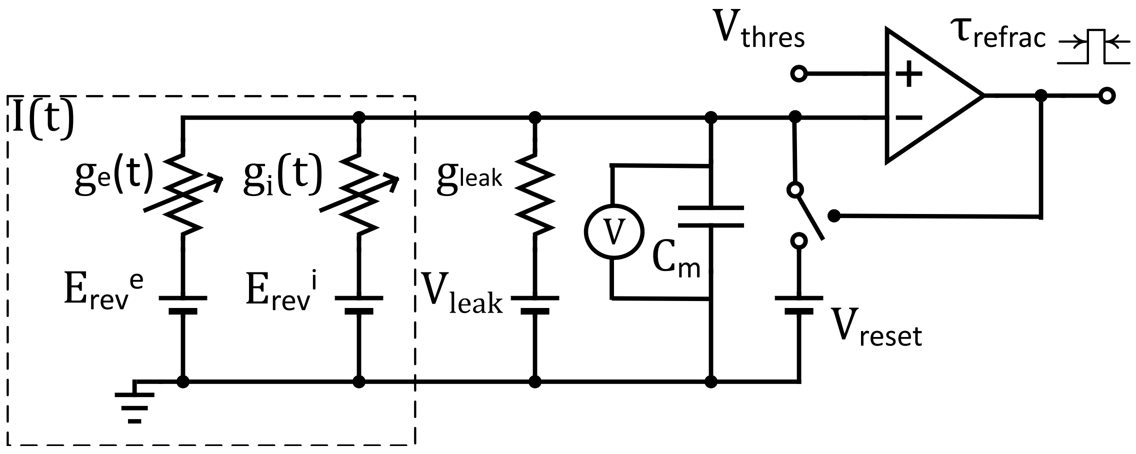


Figure 2: Conductance based circuit diagram emulating the LIF model. The synaptic current is composed of the inhibitory and excitatory synaptic currents with conductances g_c and g_i respectively. The membrane potential is emulated by the potential over the capacitor C_m. The comparator that compares the membrane potential to the threshold potential is realized via an operation amplifier (OTA), with the voltage source being short-circuited to the capacitor, forcing the membrane potential to reset to V_{reset}. τ_{refrac} denotes the refraction time, which is the time period immediately after the spike during that the neuron cannot fire again. Adapted from [Schmidt, 2024].

2.2.2 The resting potential

The resting potential V_{leak} has a directly corresponding hardware parameter V_{leak}^{CapMem}, which is stored in the capacitive memory.

2.2.3 The membrane time Constant

The membrane time constant is described by $\tau_{mem} = \frac{C_m}{g_{leak}}$, where C_m denotes the membrane capacitance, which, in the current implementation, is kept constant during calibration, and g_{leak} is the variable leak conductivity. g_{leak} is mostly, but not exclusively, controlled by the OTA's leak bias current I_{bias_leak}. That makes I_{bias_leak} the corresponding hardware parameter of the membrane time constant.

2.3 Calibration libraries

There are currently two main libraries implementing the calibration that are relevant for this thesis, namely `calibtic`, which was introduced for BrainScaleS-1, and `calix`, which was developed for BrainScaleS-2. The following sections lay out their basic functionality.

2.3.1 The original calibration module: calibtic

This section presents the former calibration library: `calibtic`. It is important to note that `calibtic` does not implement the calibration algorithm itself, but is responsible for data access and storage. There is a separate library for data acquisition, `cake`. Together, the two libraries implement a translation based parametrization approach, similar to the one developed in this thesis [Jeltsch, 2014]. Its basis is formed by a transformation class in `calibtic`, whose attributes are shown in Figure 3. The database stores calibration data in the form of parameters for certain predefined functions, i.e. polynomials [Müller et al., 2022]. This, however, only considers one dimensional dependencies [Visions, 2013], meaning each hardware parameter depends on a single model parameter. Yet in [Hinterding, 2025] it was shown, that some hardware parameters are interdependent, which makes the development of a translation-based parametrization interface with multi-parameter correlations one of the main objectives of this thesis.

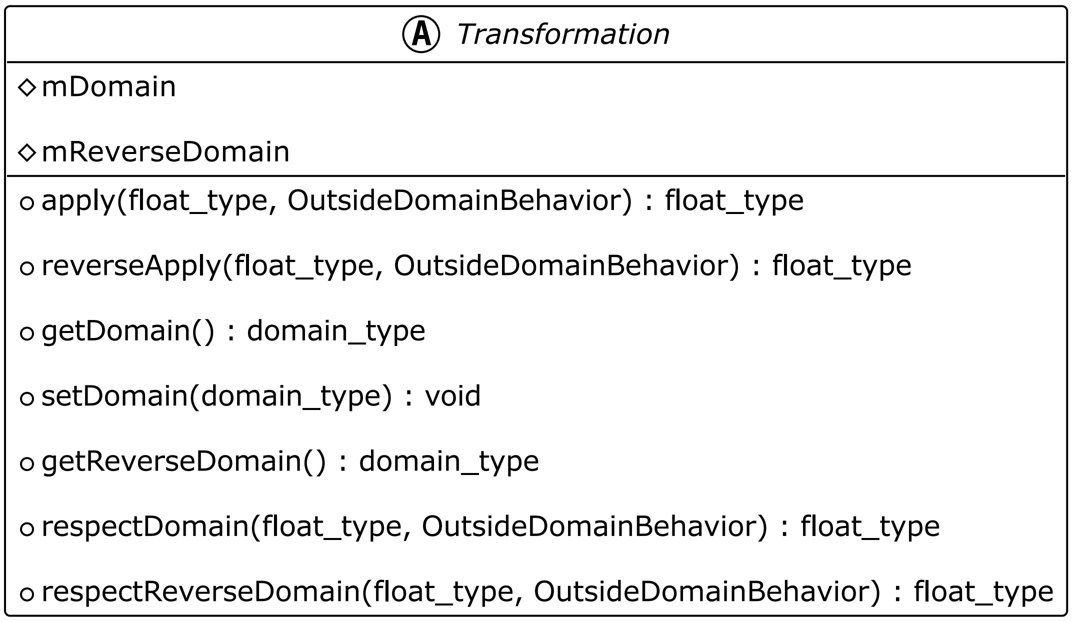

A Transformation
◇mDomain
◇mReverseDomain
o apply(float_type, OutsideDomainBehavior) : float_type
o reverseApply(float_type, OutsideDomainBehavior) : float_type
o getDomain() : domain_type
o setDomain(domain_type) : void
o getReverseDomain() : domain_type
o respectDomain(float_type, OutsideDomainBehavior) : float_type
o respectReverseDomain(float_type, OutsideDomainBehavior) : float_type

Figure 3: Abstract Transformation base class from `calibtic`. OutsideDomainBehavior is an `enum`- type that either clips the value to closest value in the domain (default), throws an exception or ignores depending on the input case. The transformation only takes one dimensional inputs. Adapted from [Visions, 2013].

2.3.2 The current calibration module: calix

`calix` was later introduced as a new calibration library [Electronic Visions, 2023]. Contrary to `calibtic`, `calix` is a data acquisition library, similar to `cake`. However, instead of a translation-based model, the calibration algorithm utilizes a bisection approach, where the hardware parameters are repeatedly updated based on a measurement of the model parameters, until the optimal hardware setting is found. To realize that, a calibration class is implemented with the methods `run()`, `prelude()`, `configure_parameters()`, `postlude()` and `measure_results`. Figure 4 shows an activity diagram visualizing an exemplary calibration algorithm used in `calix`.

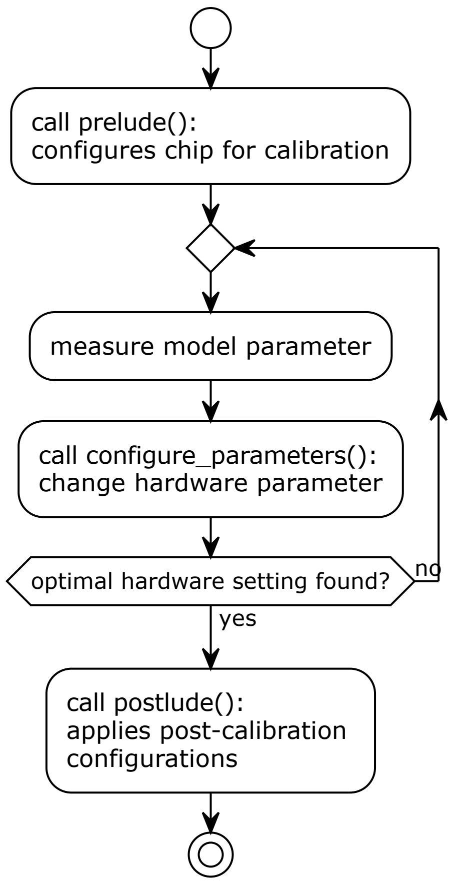

call prelude():
configures chip for calibration
measure model parameter
call configure_parameters():
change hardware parameter
optimal hardware setting found?
no
yes
call postlude():
applies post-calibration
configurations

Figure 4: Visualization of the standard calibration algorithm.

2.4 Data measurements

The developments in this thesis are based on measurements performed in [Hinterding, 2025]. The results of this work are summarized in the following section.

2.4.1 Characterization of the Analog to Digital Converters (ADC)

Prior to this work, the readout values of the respective analog-to-digital converters (ADC) determined the units of the calibration targets [Schmidt, 2024]. However, these units are not interpretable as they depend on the ADC configuration. Therefore, measurements were made in [Hinterding, 2025] to determine fitted function parameters that translate the ADC readout units into physical quanitities, volts. The results of those measurements can be seen in Figure 5.

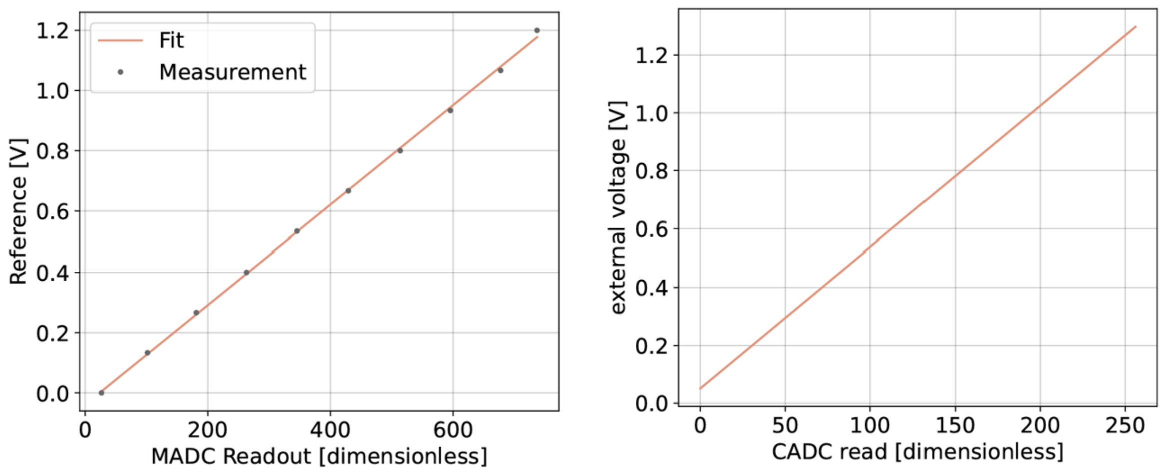


Figure 5: (a) Linear translation of the MADC readout to volts. (b) Linear translation from CADC to volts. Taken from [Hinterding, 2025]

2.4.2 2D-parameter translation

It was discovered, that some hardware parameters show a dependency on each other, specifically V_{leak}^{CapMem} and $I_{bias\_leak}^{CapMem}$. To describe this, V_{leak}^{CapMem} was measured in relation to V_{leak} for discrete $I_{bias\_leak}^{CapMem}$ values. Similarly, $I_{bias\_leak}^{CapMem}$ was measured in relation to τ_{mem} for discrete V_{leak}^{CapMem} values. Figure 6 visualizes the result.

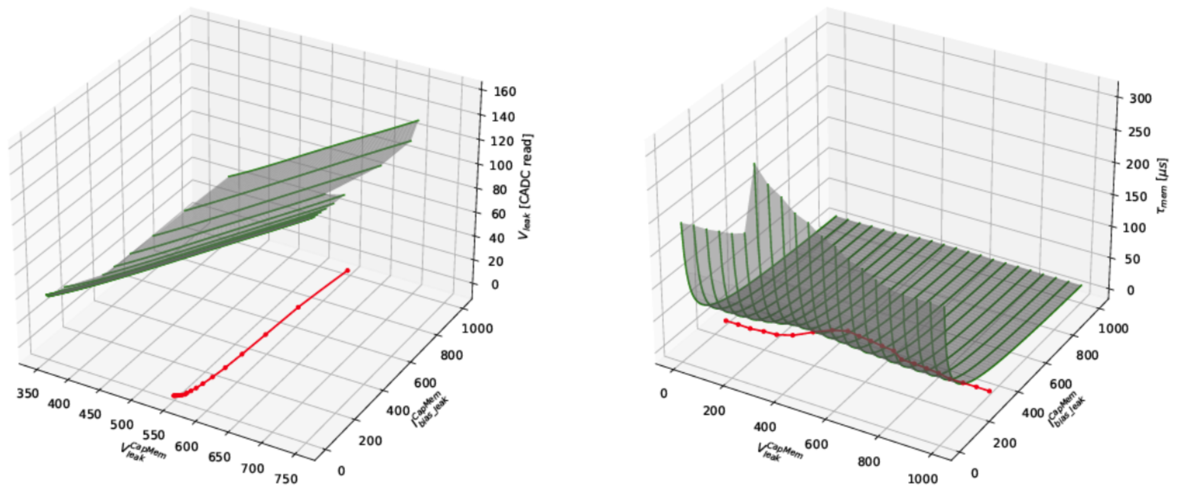


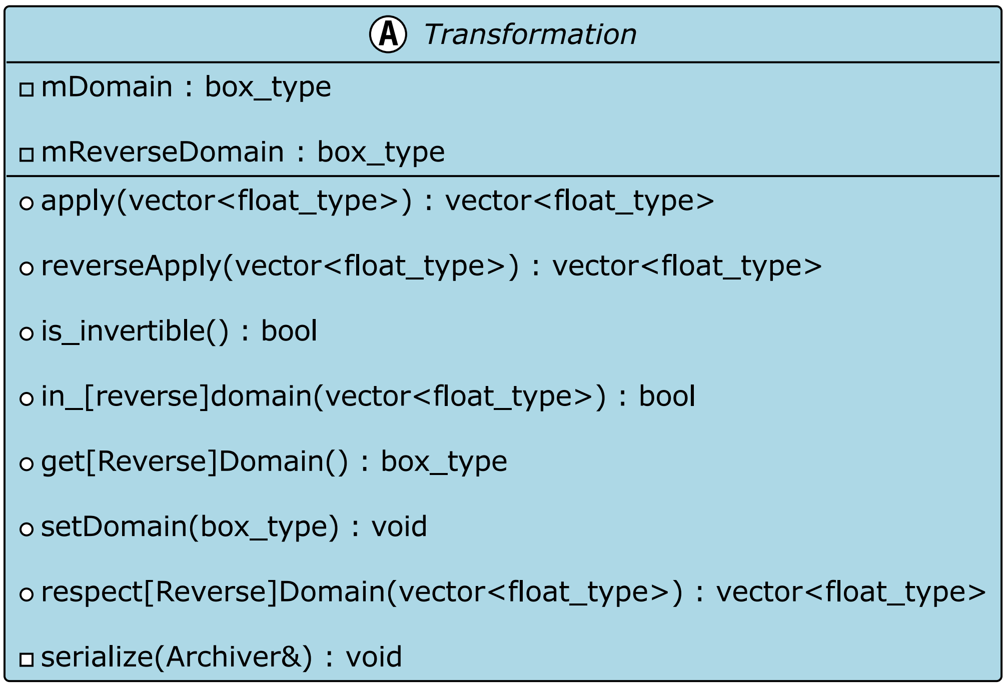
Figure 6: The left plot shows the measurements of V_{leak}^{CapMem} for discrete $I_{bias\_leak}^{CapMem}$ and the right plot shows the measurement of $I_{bias\_leak}^{CapMem}$ for discrete V_{leak}^{CapMem}. To find the $(V_{leak}^{CapMem}, I_{bias\_leak}^{CapMem})$ pair to the corresponding (V_{leak}, τ_{mem}) pair it is required to project the discrete set of functions into the $V_{leak}^{CapMem} - I_{bias\_leak}^{CapMem}$ plane and find the intersection between the resulting interpolated contour lines (red). Taken from [Hinterding, 2025]

3 Results

This section presents the key aspects of the software implementation for an ADC translation from digital units into SI units, as well as a 2D-parameter translation for hardware parameter interdependencies. The system uses $n \rightarrow m$ dimensional transformations to describe parameter dependencies. Those can then be applied to translate model- to calibrated parameters. C++ was chosen as the implementation language, as it offers shorter algorithm execution times compared to Python. The classes implemented in C++ within the scope of this work are exposed to Python using `genpybind`, a tool that generates `pybind11` bindings.

3.1 The $n \rightarrow m$ dimensional Transformation

The $n \rightarrow m$ dimensional transformations implement the abstract base class `Transformation`, which is similar to the one shown in 2.3.1 but extended to support multiple dimensions, as illustrated in Figure 7.



A Transformation
mDomain : box_type
mReverseDomain : box_type
apply(vector<float_type>) : vector<float_type>
reverseApply(vector<float_type>) : vector<float_type>
is_invertible() : bool
in_[reverse]domain(vector<float_type>) : bool
get[Reverse]Domain() : box_type
setDomain(box_type) : void
respect[Reverse]Domain(vector<float_type>) : vector<float_type>
serialize(Archiver&) : void

Figure 7: Abstract $n \rightarrow m$ dimensional Transformation class structure. `apply()` and `reverseApply()` take mutltidimensional inputs. The domain is represented by `box_type`, which is a vector of intervals. `float_type` denotes a `double` in the given case, can however be changed to any float-like type. Note that the diagram only shows the methods relevant for illustrating the main functionality.

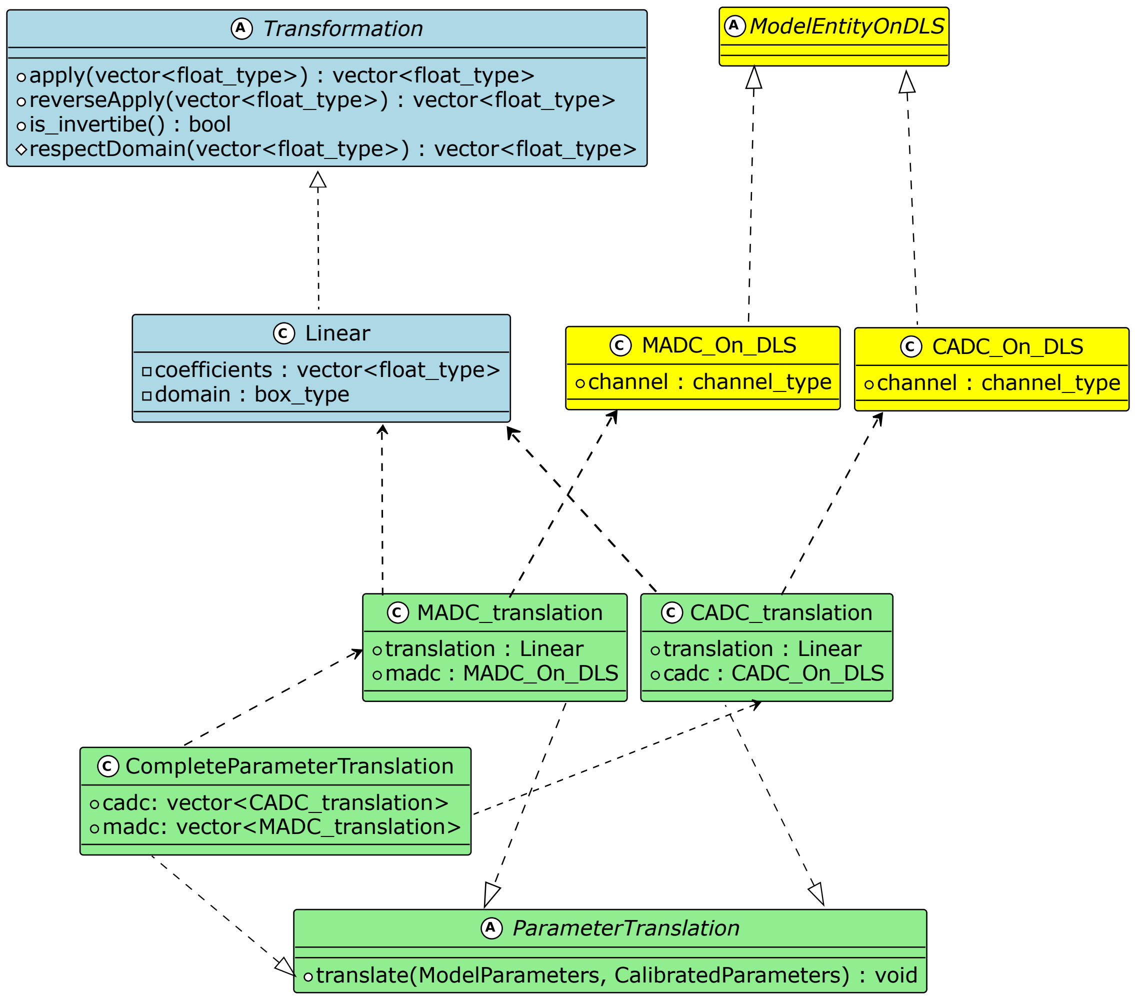
In contrast to the calibtic equivalent the `apply()` and `reverseApply()` methods take vector inputs. `mDomain` signifies the outer dimensionality, which must be of a multi-dimensional box-type. This is implemented using a vector of intervals that contains a domain for every dimension. The reverse domain is then computed by applying the transformation to the domain limits. Furthermore, the invertability of a multi-dimensional function is not trivial and therefore needs to be checked in `is_invertible()`.

3.2 ADC translation

This section presents the development of a system architecture that enables the translation of ADC readout units into physical quantities. Parts of this architecture are also used in the development of a two-dimensional parameter translation described in later sections. It should be noted that only neuron membrane measurements are considered here.

3.2.1 Parameter Translation

As described in [Hinterding, 2025], the ADC to SI unit translation has a linear shape. Therefore, the new ADC characterization developed in the scope of this thesis implements the abstract `Transformation` class for the simplest possible case: the linear transformation. Even though the ADC only takes a one-dimensional input signal, the general structure of the `Linear` class allows arbitrary input dimensionality for reasons discussed in Section 4.
As every parameter generally has different dependencies and therefore requires different translation functions, an abstract type `ParameterTranslation` was created, which allows every parameter to contain its own translation attribute. The derived translation classes (i.e. `MADC_translation` and `CADC_translation`) implement the virtual `translate()` method depending on what is expected as the result of the respective translation. For the ADC case `translate()` simply yields the transformation rather than the translated parameter, as the ADC translation needs to be invertible. The `CompleteParameterTranslation` class implements `translate()` such that it calls the `translate()` methods of all the parameter translation classes requested by the user, so that all translations can be performed with one function call. This hierarchy is visualized in Figure 8.



Transformation
apply(vector<float_type>) : vector<float_type>
reverseApply(vector<float_type>) : vector<float_type>
is_invertibe() : bool
respectDomain(vector<float_type>) : vector<float_type>
ModelEntityOnDLS
Linear
coefficients : vector<float_type>
domain : box_type
MADC_On_DLS
channel : channel_type
CADC_On_DLS
channel : channel_type
MADC_translation
translation : Linear
madc : MADC_On_DLS
CADC_translation
translation : Linear
cadc : CADC_On_DLS
CompleteParameterTranslation
cadc: vector<CADC_translation>
madc: vector<MADC_translation>
ParameterTranslation
translate(ModelParameters, CalibratedParameters) : void

Figure 8: Class hierarchy of the ADC characterization. `Linear` implements `Transformation` and is a `MADC_translation` and `CADC_translation`. `ParameterTranslation` forms the abstract basis of all parameter translations and `CompleteParameterTranslation` defines the `translate()` method that translates all requested parameters. Note that the translation functions differ not only between CADC and MADC but also between their respective channels, which is why `CADC_translation` and `MADC_translation` each hold a member derived from `ModelEntityOnDLS`, which specifies the channel. All classes implement a data serialization interface.

Notably, `translate()` does not have a return value, but rather takes two input arguments, `ModelParameters` and `CalibratedParameters`, that are both polymorphic maps and the latter of which is modified during the function runtime. `ModelParameters` contains the parameters for which a translation is requested and `CalibratedParameters` contains the results of the calibration. In `CompleteParameterTranslation`, `translate()` is generally called with `CalibratedParameters` being empty. The method then iterates through the translations, and in turn calls their `translate()` methods, that verify whether the parameter is requested in `ModelParameters`. If so, the calibrated parameter is added to the `CalibratedParameters` map. Figure 9 visualizes this algorithm.
The keys of both maps are of type `ModelEntityOnDLS`, which represents the model enti-

ties on the chip, i.e. CADC or MADC. As previously mentioned, the keys are polymorphic, which cannot be realized using `std::unordered_map <typename Key, typename Value>`, therefore, a custom map is used to support this functionality. This requires the keys to be hashable, which is why all classes derived from `ModelEntityOnDLS` implement a `hash()` method.

For MADC and CADC, the translation functions differ between channels, so `MADC_On_DLS` and `CADC_On_DLS` store the specific channel for which the respective translation is requested. This allows for arbitrary channel configurations.

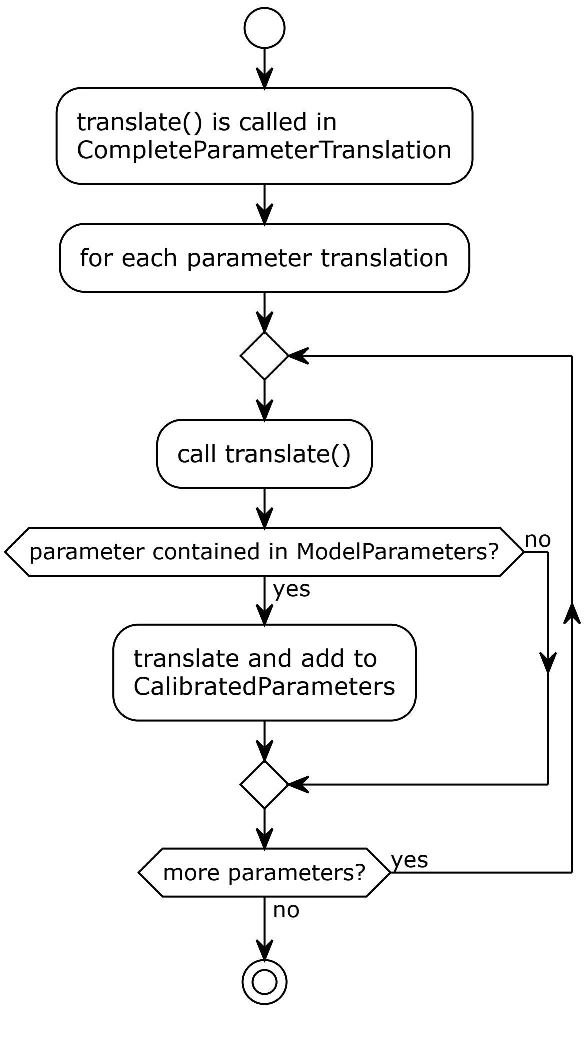

translate() is called in CompleteParameterTranslation
for each parameter translation
call translate()
parameter contained in ModelParameters?
no
yes
translate and add to CalibratedParameters
more parameters?
yes
no

Figure 9: Schematic visualizing the translation algorithm. The `iterate` method in `CompleteParameterTranslation` iterates through all parameter translations and calls their respective `translate()` methods. These verify whether the parameter is contained in `ModelParameters`, if so, the calibrated parameter is added to `CalibratedParameters`.

3.2.2 Integration into the calibration system

The architecture laid out above was integrated into the existing calibration module in PyNN, which makes it accessible to the user. When conducting an experiment, a custom parameter translation can be constructed and passed to the setup. An example construction can be seen in Listing 1.

```python
targets = pyccalix.ParameterTranslation.ModelParameters()
#example coefficients for Linear trafo (MADC)
coeff_madc = [[0.0018685445400704107],[-0.43310387776092285]]
#create Linear transformation objects
LinearTrafoMADC_1 = pyccalix.Linear(coeff_madc)
#specify channels
madc_config = halco.SourceMultiplexerOnReadoutSourceSelection(0)
#create targets
targets.set(pyccalix.MADC_On_DLS(madc_config), pyccalix.ModelMADC
    ())
#create translation
translation = pyccalix.MADC_translation(pyccalix.MADC_On_DLS(
    madc_config), LinearTrafoMADC_1)

pynn.setup(injected_parameter_translation = pyccalix.
    CompleteParameterTranslation([],
      [translation]))
```

Listing 1: Example construction of a translation of ADC values for measurements of the membrane capacitance in Python.

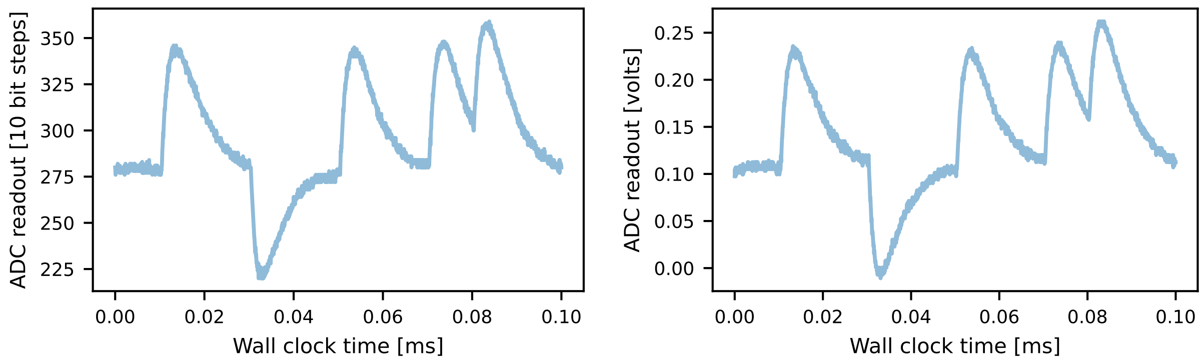
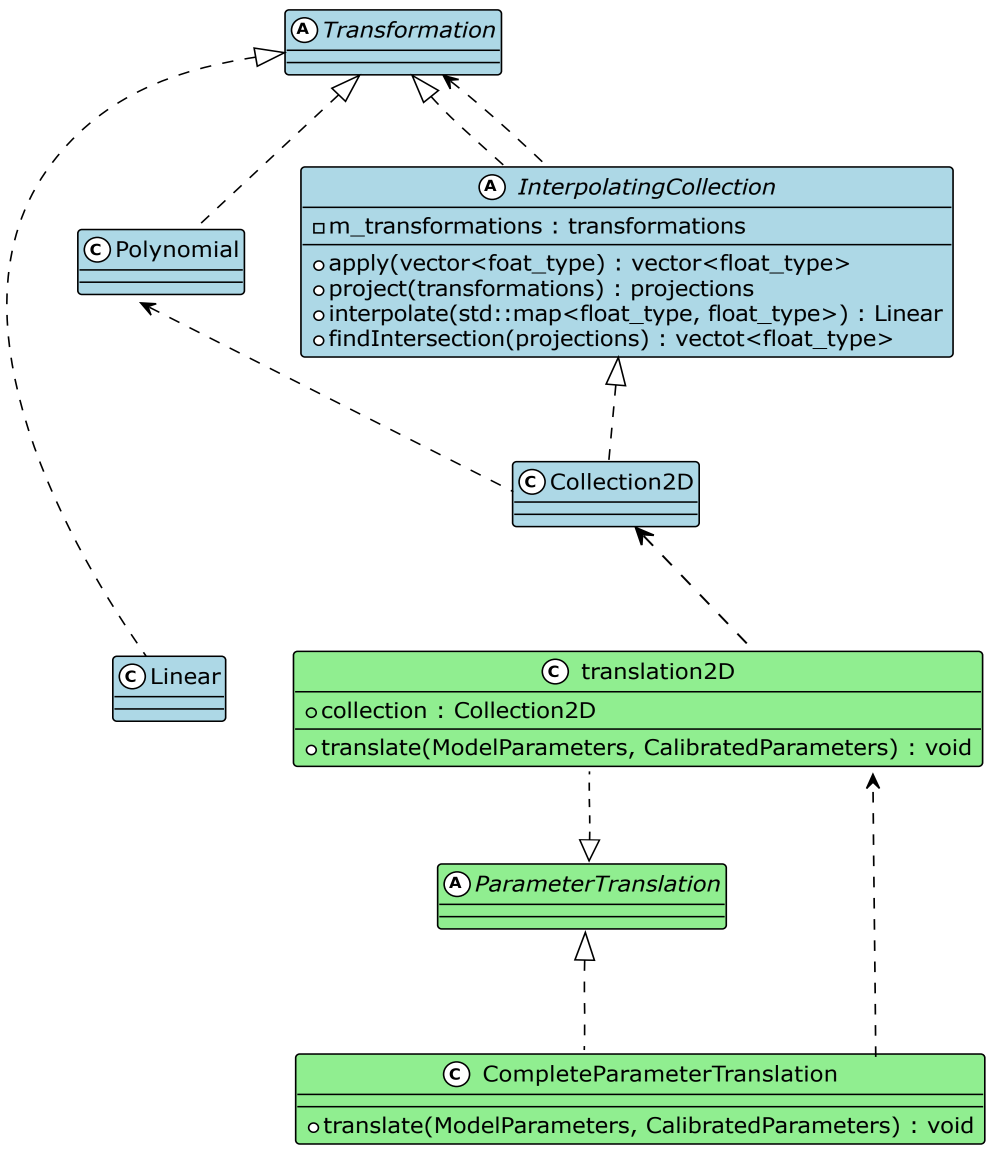


Figure 10: ADC readout in (left) digital steps and (right) volts during a single neuron experiment. External stimulation was applied.

3.3 Multi-parameter dependencies

3.3.1 Class structure and algorithm

Moving on from the ADC characterization, the next objective was to enable the translation-based parametrization of hardware parameters that depend not only on the desired model parameters but also on each other. Specific parameters for which such dependencies have been measured are V_{leak}^{CapMem} and $I_{bias\_leak}^{CapMem}$. The goal was to develop a software implementation that allows the calibration of these parameters using the measurements laid out in [Hinterding, 2025]. A class diagram visualizing the systems structure is shown in Figure 11.



Transformation
InterpolatingCollection
m_transformations : transformations
apply(vector<foat_type) : vector<float_type>
project(transformations) : projections
interpolate(std::map<float_type, float_type>) : Linear
findIntersection(projections) : vectot<float_type>
Polynomial
Collection2D
Linear
translation2D
collection : Collection2D
translate(ModelParameters, CalibratedParameters) : void
ParameterTranslation
CompleteParameterTranslation
translate(ModelParameters, CalibratedParameters) : void

Figure 11: Class diagram showing the basic structure of the 2D-translation. `InterpolatingCollection` implements `Transformation` and holds discrete sets of transformations at different positiions. `apply()` projects these transformation collections into n-1 dimensional space for specific input values (model parameters) and finds the intersection between the interpolated contour lines resulting from the different sets. The intersection is equivalent to the corresponding set of calibrated parameters. `Collection2D` implements `InterpolatingCollection` for two dimensions. Its member collection consists of sets of polynomials.

The foundation for this translation is formed by the `InterpolatingCollection` abstract class, which is derived from `Transformation`. `InterpolatingCollection` contains all collections of inverted transformations at discrete positions in an n-dimensional space as a polymorphic map. In this case `apply()` takes an input vector containing the values of the model parameters and returns a vector representing the set of corresponding hardware parameters. `apply`, in turn, calls the `project()` method, which returns the set of projected points in n-1 dimensional space, by applying the inverted transformations at the given model parameter values. An interpolation is then applied to find the intersection between the resulting planes, which is equivalent to the set of to-hardware transformed model parameters.

While `apply()` and `project()` are implemented for arbitrary dimensionality, the `interpolate()` and `findIntersection()` methods are pure virtual and thus require an implementation in the derived classes. This approach was chosen for simplification, as two dimensions are sufficient for the case at hand, though ideas for a generalised interpolation algorithm are still going to be discussed in the Section 4.

`Collection2D` implements `InterpolatingCollection` for two dimensions. It contains two collections of polynomials, each positioned at discrete values along the one-dimensional hardware parameter axes. For a given model parameter, the points where the transformation equals the model parameter value are projected, as shown in Figure 6. The goal is then to determine the point where the interpolations through the two sets of points intersect.

Piecewise linear interpolation was chosen because it is computationally efficient and the functions are assumed to be linear to a sufficient degree [Hinterding, 2025]. To find the intersection, a brute-force approach was initially considered, in which the interpolation is applied to each pair of neighboring points in one set and intersected with the interpolation applied to each pair of neighboring points in the other set. However, this approach scales as $O(n^2)$ on average, which is not optimal. Therefore, an alternative algorithm was developed that takes advantage of the fact that the two sets are sorted along their respective axes. It is also reasonable to assume that, as long as the correlation is weak, the points closest to each other are also closest to the intersection. An activity diagram illustrating this algorithm is shown in Figure 12.

After applying the projection, a binary search is performed for each point in both sets to find the closest point from the other set along the coordinate axis by which the set is sorted. This produces two containers of the closest pairs along each axis. The pair with the smallest absolute distance across both sets is considered the closest pair. Linear interpolation is then applied between the points of the closest pair and their neighboring points. If no intersection is found within the given bounds, the points of the closest pair are removed from the projections, and the process repeats until either an intersection is found or no projected points remain, in which case `findIntersection()` returns an empty vector.

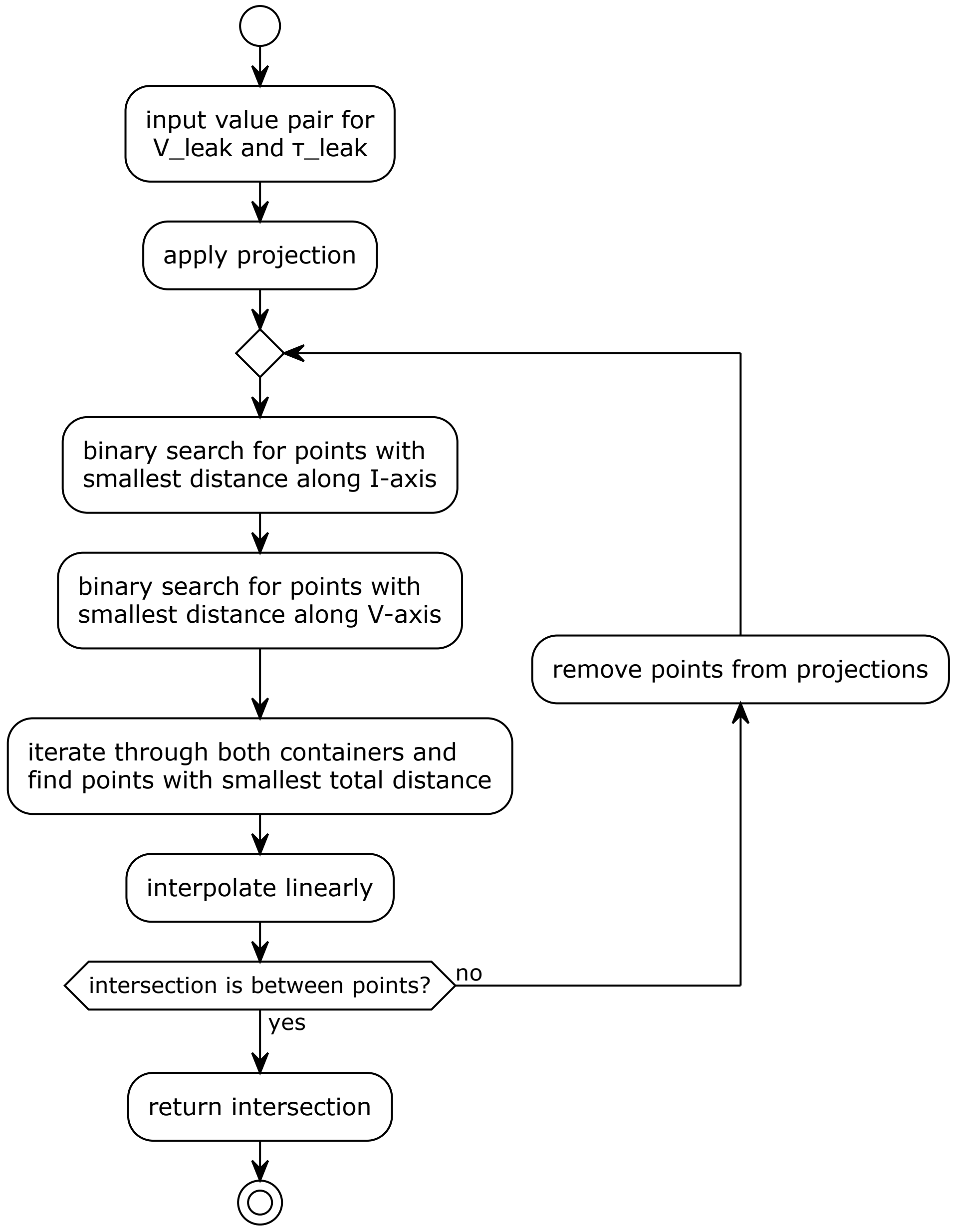

input value pair for V_leak and τ_leak
apply projection
binary search for points with smallest distance along I-axis
binary search for points with smallest distance along V-axis
iterate through both containers and find points with smallest total distance
interpolate linearly
intersection is between points?
no
yes
return intersection
remove points from projections

Figure 12: Activity diagram visualizing the algorithm for finding the intersection. The projection produces two discrete sets of points, each sorted along one of the coordinate axes. Using binary search, the closest point along the relevant axis is found for each point in each set. Iterating through the resulting pairs, the pair with the smallest absolute distance is identified. A linear interpolation is then applied between these points and their neighbors. If the intersection lies between the points, the search stops; otherwise, the points are removed, and the process repeats until an intersection is found or no points remain. The algorithm has an average-case runtime of $O(n \log n)$, assuming weak correlations.

Notably, this algorithm has a worst-case runtime of $O(n^2)$, which occurs if the intersection lies between the points that are furthest apart or if no intersection is found. The former case, however, is considered negligible, as the correlation between the parameters is assumed to be weak in most instances. On average, the algorithm scales as $O(n \log n)$ due to the efficiency of binary search, vastly outperforming the brute-force approach for large data sets.

Following the structure shown in Figure 8, the ADC-characterization section `translation2D` is derived from the `ParameterTranslation` class. The `translate()` method applies the `Collection2D` transformation to the given model parameters and adds a calibrated parameter containing the result to the `CalibratedParameters` map.

3.3.2 Runtime analysis

This section presents a runtime evaluation of the calibration algorithm presented above compared to one the shown in Figure 4. For that, the measurement result is extrapolated to the number of analog neuron parameters on the chip. Since there are 512 neurons and 27 analog neuron parameters, the number of parameters per chip sums up to 13824. For this analysis, it is distinguished between the best and worst case. The best case considers only linear transformations for all parameters, while the worst case that is realizable with the architecture developed in this work assumes that every parameter is calibrated using a 2D-collection of polynomials. For both cases, the measurement was repeated five times, then the average was taken and the error was determined using the gaussian approximation.

First, it is assumed that all parameters are calibrated using a linear transformation. In this case, a loop with 13,824 iterations was executed, since the time required for a single transformation was too short to yield a sufficiently accurate measurement. The measured execution time is approximately 15.79 ± 0.01ms.

Using only collections of polynomials the estimated calibration time would be (35.2 ± 5.5)s. This was approximated by running a loop over 9912 collections, one for each pair of parameters, where the polynomials has different offset values, but the same number of terms in each collection. The input values of `apply()` were also slightly varied between each call. In this case, an intersection was always found in the first iteration, which is the best case.

If the input values were chosen such that an intersection was never found, the runtime increased approximately by order of three, since the algorithm then scales with $O(n^2)$. In praxis, this would, however, only ever be the case when invalid model parameters are requested.

In contrast, the current single operation-point calibration takes 7.46min to calibrate all parameters on the chip. This order of magnitude improvement is a promising result for faster calibration processes in the future.

3.3.3 Memory footprint analysis

This section evaluates the memory footprint of the data structures presented in this work. The footprint was approximated by summing the memory usage of all attributes in the respective classes. The results are summarized in Table 1. For `Polynomial1D`, the footprint depends on how many terms the polynomial has, as the implementation features two maps containing the coefficient and offset for every term. The progression of the footprint depending on the number of terms in the polynomial is shown in Figure

13. `Collection2D` was approximated by taking example data from [Hinterding, 2025]. However, the size of `Collection2D` linearly grows with the number of polynomials it stores. The footprint of `CompleteParameterTranslation` was determined for an MADC translation vector of size 2, a CADC translation vector of size 1024 and a `Collection2D` for V_{leak} and τ_{mem}.

Data type	**Approximated memory footprint (kilobytes)**
`Linear`	0.20
`Polynomial1D`	0.94
`Collection2D`	26
`vleak_taumem_translation`	26
`CADC_translation`	0.21
`MADC_translation`	0.21
`CompleteParameterTranslation`	231.2

Table 1: Approximated memory footprint of the relevant data types developed in this work. Note that the values for `Polynomial1D` and `Collection2D` are exemplary and may vary depending on factors such as the number of polynomial terms or the number of polynomials in the collection. The `Linear` type is assumed to be one-dimensional, as it is in the ADC translations.

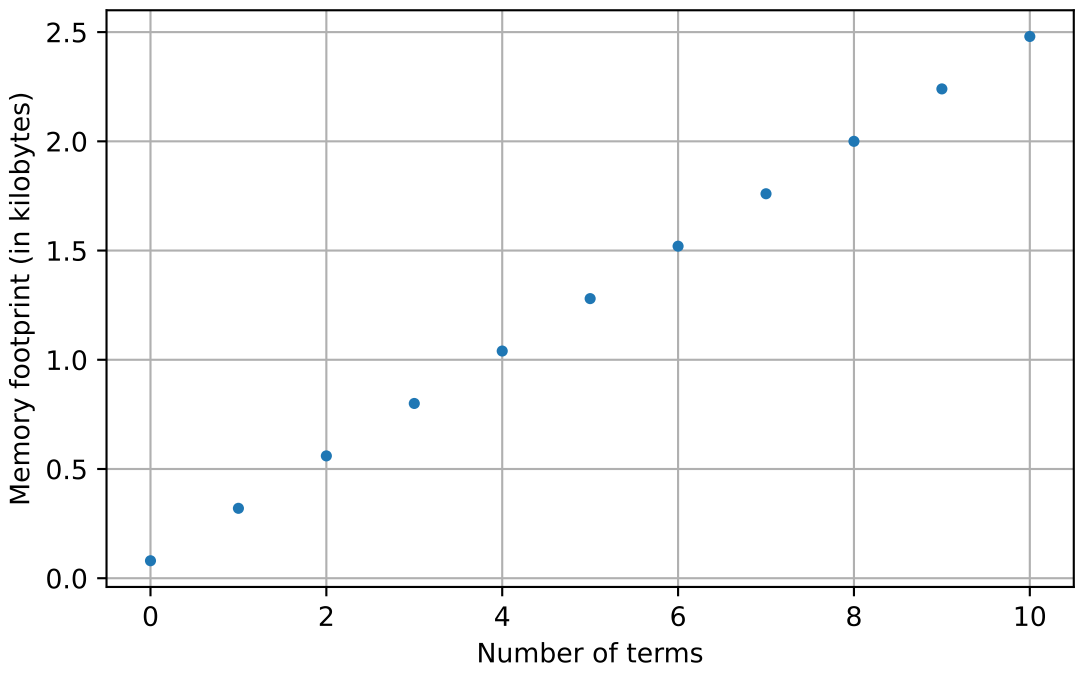


Figure 13: Plot showing the linear progression of the polynomials memory footprint against the number of terms. The maximum number was chosen to be 10 because in the data from [Hinterding, 2025] the polynomials do not feature more terms.

4 Summary and outlook

In this work, a software architecture was developed, that enables the translation-based parametrization of BSS-2. This architecture builds on transformations developed for the calibration library of BrainScaleS-1. However, these are extended to allow multiple hardware parameter interdependencies. This was realized by implementing multi-dimensional transformations. A class hierarchy was developed that enables the calibration of all parameters requested by the user with one function call.

As a first step, an ADC translation was added to the calibration library, that allows to convert digitized ADC values to volts. This makes the ADC readouts better interpretable, since, unlike unitless values, SI-units are independent of the ADC setup. This translation was then integrated into the high-level SNN experiment definition language PyNN, where it can be accessed by users.

Next, a translation was implemented that handles interdependencies of two hardware parameters. For finding the corresponding pair of hardware parameters to a pair of model parameters, an algorithm with an average runtime of $nlog(n)$ was developed.

It was shown that a translation-based algorithm achieves a significantly faster runtime than the single fixed-point approach. Nevertheless, a translation-based parametrization can only provide reasonable results in terms of time and resource usage if hardware parameter interdependencies are low-dimensional. Moreover, as discussed in Section 3.3.3, the memory footprint of large collections of complex functions becomes substantial.

While the parametrization accounts for 2D-parameter dependencies, in the future there might be instances where more than two interdependent parameters will have to be considered.
As mentioned in Section 3.3.1, the algorithm finding the set of hardware parameters for a set of model parameters is only partly implemented for arbitrary dimensionality. Specifically the part where an interpolation would need to be found for points in n-dimensional space it was decided to adhere to two dimensions, with this being the relevant case at this point. However, concepts were explored for a general solution.
Perhaps the simplest model would be an extrapolation of the part-wise bilinear interpolation to more than two dimensions. Same as with the linear interpolation in the two dimensional case, to avoid a suboptimal scaling behavior, an area where the intersection would be most probable has to be found. Similar to the approach in Section 3.3.1, a closest neighbor search would be reasonable. Possible solutions would be to organize the point clouds into k-d trees or bounding volume hierarchies to search for the points with the closest distance to the intersection area. Also, the Gilbert–Johnson–Keerthi distance algorithm [Ong and Gilbert, 1997] could be utilized to find the closest distance between the sets.

For higher-dimensional parameter dependencies, the translation-based approach is only practical to a limited extent, as the time and resource requirements for data acquisition could become prohibitive. Nevertheless, it could still be used, for example, to identify a starting point for the iterative calibration, thereby reducing the overall time requirement. Determining the true dimensionality of parameter dependencies requires further measurements. The class hierarchy developed in this work facilitates the addition of new parameter translations as needed.

Another question to consider is whether piecewise linear interpolation between the projections is reasonable in all cases. If the resolution of the point set is too low, linear interpolation may become considerably inaccurate. In such cases, spline or polynomial interpolation could be used as alternatives; however, finding intersections algorithmically in this context would be challenging. This situation is considered unlikely, as the resolution is determined by the measurement intervals.

In summary, the translation-based parametrization was shown to be faster than the current iterative model, however the data required for a full parametrization of the chip using this model has not been acquired yet. Furthermore, the dimensionality of general parameter interdependencies needs to be investigated. For that, a first step would be to compare the accuracy of the translation-based parametrization with the iterative approach for the already acquired data, as the iterative calibration implicitly considers forward dependencies.

References

[Balasubramanian, 2021] Balasubramanian, V. (2021). Brain power. *Proceedings of the National Academy of Sciences of the United States of America*, 118.

[Bouvier et al., 2019] Bouvier, M., Valentian, A., Mesquida, T., Rummens, F., Reyboz, M., Vianello, E., and Beigne, E. (2019). Spiking neural networks hardware implementations and challenges: A survey. *ACM Journal on Emerging Technologies in Computing Systems*, 15(2):1–35.

[Brette R, 2005] Brette R, G. W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J Neurophysiol*, Nov 2005.

[Electronic Visions, 2023] Electronic Visions (2023). calix. `https://github.com/electronicvisions/calix`. GitHub repository. Accessed: 2025-08-15.

[Hinterding, 2025] Hinterding, R. (2025). Towards a multidimensional calibration of neuromorphic hardware using a parameter transformation model. Bachelorarbeit, Universität Heidelberg.

[Jegham et al., 2025] Jegham, N., Abdelatti, M., Elmoubarki, L., and Hendawi, A. (2025). How hungry is ai? benchmarking energy, water, and carbon footprint of llm inference.

[Jeltsch, 2014] Jeltsch, S. (2014). *A Scalable Workflow for a Configurable Neuromorphic Platform*. PhD thesis, Universität Heidelberg.

[Müller et al., 2022] Müller, E., Schmitt, S., Mauch, C., Billaudelle, S., Grübl, A., Güttler, M., Husmann, D., Ilmberger, J., Jeltsch, S., Kaiser, J., Klähn, J., Kleider, M., Koke, C., Montes, J., Müller, P., Partzsch, J., Passenberg, F., Schmidt, H., Vogginger, B., Weidner, J., Mayr, C., and Schemmel, J. (2022). The operating system of the neuromorphic brainscales-1 system. *Neurocomputing*, 501:790–810.

[Ong and Gilbert, 1997] Ong, C. J. and Gilbert, E. (1997). The gilbert-johnson-keerthi distance algorithm: a fast version for incremental motions. In *Proceedings of International Conference on Robotics and Automation*, volume 2, pages 1183–1189 vol.2.

[Pehle et al., 2022] Pehle, C., Billaudelle, S., Cramer, B., Kaiser, J., Schreiber, K., Stradmann, Y., Weis, J., Leibfried, A., Müller, E., and Schemmel, J. (2022). The brainscales-2 accelerated neuromorphic system with hybrid plasticity. *Frontiers in Neuroscience*, Volume 16 - 2022.

[Schilling et al., 2020] Schilling, K., Werrlich, S., Thiel, F., and Harren, H. (2020). Deep learning — a first meta-survey of selected reviews across scientific disciplines, their commonalities, challenges and research impact. *arXiv preprint arXiv:2011.08184*.

[Schmidt, 2024] Schmidt, H. (2024). *Large-Scale Experiments on Wafer-Scale Neuromorphic Hardware*. PhD thesis, Universität Heidelberg.

[Visions, 2013] Visions, E. (2013). calibtic. `https://github.com/electronicvisions/calibtic`. GitHub repository. Accessed 2025-08-14.

[Weis, 2020] Weis, J. (2020). Inference with artificial neural networks on neuromorphic hardware. Master's thesis, Universität Heidelberg.

[Wilhelm et al., 2025] Wilhelm, P., Wittkopp, T., and Kao, O. (2025). Beyond test-time compute strategies: Advocating energy-per-token in llm inference. In *Proceedings of the 5th Workshop on Machine Learning and Systems*, EuroMLSys '25, page 208–215, New York, NY, USA. Association for Computing Machinery.

[Wunderlich et al., 2019] Wunderlich, T., Kungl, A. F., Müller, E., Hartel, A., Stradmann, Y., Aamir, S. A., Grübl, A., Heimbrecht, A., Schreiber, K., Stöckel, D., Pehle, C., Billaudelle, S., Kiene, G., Mauch, C., Schemmel, J., Meier, K., and Petrovici, M. A. (2019). Demonstrating advantages of neuromorphic computation: A pilot study. *Frontiers in Neuroscience*, Volume 13 - 2019.

Funding acknowledgement

The work carried out in this Bachelor Thesis used systems, which received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Nos. 720270, 785907 and 945539 (Human Brain Project, HBP) and Horizon Europe grant agreement No. 101147319 (EBRAINS 2.0)

Reproducibility statement

The results of this work can be reproduced using the repositories listed in Table 2.

What?	**Name**	**Reference**	**Note**
Repository	waf	cef18ea	Upstream
Repository	singularity	898e91a	Upstream
Repository	calix	59c478e	In Review (25245)
Repository	pynn-brainscales	d2b0345	In Review (25086)
Reporitory	hate	I5337f	Upstream
Repository	halco	8129637	Upstream
Container	stable container	$2025-07-13_1$.img	-

Table 2: Repositories and containers required for the result reproduction.

Acknowledgements

I would like to thank the whole of the Electronic Visions group for creating the most welcoming environment and allowing me to take part in their work. Special thanks goes to Eric and Yannik for always helping me out, as well as for proof reading my thesis. Also special thanks to David and Tim for making me look forward to come to the office, and for also proof reading my thesis. I would further like to thank Philipp for the best mentoring I could hope for. Thanks goes also to Prof Johannes Schemmel and Prof Jürgen Hesser for being my first and second examinators.

I am grateful to my friends and family, especially Sebastian, who supported me a lot during the time of writing, and my father, for getting me into programming in the first place.

Lastly, I would like to thank Eva Oettinger, my high school physics teacher, without whose enthusiastic teaching and motivation I would have never pursued physics.

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 21.08.2025,

[illegible]